Component-Based Analysis of Fault-Tolerant Real-Time Programs[*]

Borzoo Bonakdarpour[†] Sandeep S. Kulkarni[†] Anish Arora[§]

[†]Department of Computer Science and Engineering
Michigan State University
East Lansing MI 48824 USA
Email: {borzoo, sandeep}@cse.msu.edu
http://www.cse.msu.edu/∼{borzoo, sandeep}

[§]Department of Computer Science and Engineering
Ohio State University
Columbus Ohio 43210 USA
Email: anish@cse.ohio-state.edu
http://www.cse.ohio-state.edu/∼anish

Abstract

We focus on decomposition of fault-tolerant real-time programs that are designed from their fault-intolerant versions. Towards this end, motivated by the concepts of *state predicate detection* and *state predicate correction* [1] for *untimed* systems, we identify three types of components, namely, *detectors*, *weak δ-correctors*, and *strong δ-correctors*. We also consider different levels of fault-tolerance, namely, *soft-failsafe, hard-failsafe, nonmasking, soft-masking*, and *hard-masking*, depending upon the satisfaction of safety, liveness, and timing constraints in the presence of faults. We show that depending upon the level of tolerance, fault-tolerant real-time programs contain one or more detectors and/or weak/strong-δ correctors.

Keywords: Fault-tolerance, Real-time, Component-based design, Decomposition, Bounded-time recovery Formal methods.

1 Introduction

We analyze real-time fault-tolerant programs that are designed from their fault-intolerant versions. Such fault-tolerant programs may be designed for maintenance to deal with a previously unanticipated class of faults or for separating functionality of the program from its tolerance properties. In both these cases, "reuse" of existing program is desirable, and possibly mandatory. An important concern for reuse-based techniques for design of fault-tolerance is their *completeness*. Intuitively, completeness of a reuse-based technique captures the ability of that technique to produce any fault-tolerant program from a fault-intolerant program, say p, assuming that there is some reuse-based design of a fault-tolerant version of p. Said another way, if p can be made fault-tolerant by any reuse design, then the technique should be able to yield one such fault-tolerant program. Our

[*]This work was partially sponsored by NSF CAREER CCR-0092724, DARPA Grant OSURS01-C-1901, ONR Grant N00014-01-1-0744, NSF grant EIA-0130724, and a grant from Michigan State University.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **2007**	2. REPORT TYPE	3. DATES COVERED **00-00-2007 to 00-00-2007**
4. TITLE AND SUBTITLE **Component-Based Analysis of Fault-Tolerant Real-Time Programs**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Michigan State University ,Department of Computer Science and Engineering,East Lansing,MI,48824**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES **http://www.cse.msu.edu/publications/tech/TR/MSU-CSE-07-24.pdf**		
14. ABSTRACT		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**25**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

focus in this paper is on the completeness of reuse-based techniques in the context of component-based design of fault-tolerant real-time programs.

Regarding completeness, there are two main issues in such component-based design: (1) *Design method*: where one focuses on transforming the given program into a fault-tolerant program, and (2) *Containment question*: where we want to determine whether such components exist in fault-tolerant programs irrespective of how they are designed. Regarding the first issue, previously, in [1], Arora and Kulkarni have presented a sound and complete method for component-based design of fault-tolerant *untimed* programs. Their method is based on the principle of *state detection* and *state correction*. In particular, using this principle, they identify two components, namely, *detectors* and *correctors* that respectively focus on detecting whether the execution of a given program action is safe in the given state and restoring the execution of a program to a state where a certain state predicate is true. They subsequently show that (1) detectors are the necessary and sufficient building-block for designing *failsafe* fault-tolerant programs, i.e., programs that satisfy their safety specification in the presence of faults, (2) correctors are necessary and sufficient building blocks for designing *nonmasking* programs, i.e., programs that recover to legitimate states after occurrence of faults, and (3) both are necessary and sufficient for masking programs, i.e. programs where both safety and liveness specification are satisfied in the presence of faults.

In this paper, we focus on the second question. In particular, we investigate whether the idea of state detection and correction can be applied to real-time fault-tolerant programs. Towards this end, we define three types of components, namely, *detectors*, *weak δ-correctors*, and *strong δ-correctors*. Similar to [1], detectors are based on the concept of detecting *state predicates*. However, based on the closure properties of the correction state predicate, we introduce *weak* and *strong* δ-correctors.

We illustrate that depending upon the level of fault-tolerance, existing real-time fault-tolerant programs that reuse their fault-intolerant version contain one or more of the above components. We show the existence of the components by investigating the necessary conditions under which fault-tolerance can be provided in the context of real-time programs. Towards this end, we refine the levels of fault-tolerance considered in [1] based on the satisfaction of safety, liveness, and *timing requirements* (e.g., deadlines) in the presence of faults. In particular, we consider *soft-failsafe*, *hard-failsafe*, *nonmasking*, *soft-masking*, and *hard-masking* fault-tolerance (cf. Section 2 for precise definitions). Intuitively, a *soft* fault-tolerant program is not required to meet its timing constraints in the presence of faults. However, in the absence of faults a soft fault-tolerant program behaves like its fault-intolerant version, i.e., it satisfies its timing constraints in the absence of faults. On the other hand, a *hard* fault-tolerant program must satisfy its timing constraints even in the presence of faults. In other words, in hard fault-tolerant programs, the demand for hard real-time processing merges with catastrophic consequences of systems, whereas in soft fault-tolerance the catastrophic consequences are not related to the program's timing constraints. Furthermore, for nonmasking, soft-masking, and hard-masking levels of fault-tolerance, we distinguish two cases where state correction is achieved in bounded or unbounded amount of time. In other words, we consider *unbounded* and *bounded-time recovery* in the presence of faults.

In order to formally express timing constraints of a real-time program, we focus on a standard property in real-time systems called *bounded response*. Intuitively, a bounded response property specifies if a state predicate, say P, becomes true then another state predicate, say Q, must become true within a bounded amount of time, say θ. Observe that according to [2, 3], such properties fall in the category of safety properties. Thus, in this paper the notion safety specification consists of a timed part (i.e., a set of bounded response properties) and an untimed part (e.g., state perturbations), which is modeled by a set of bad transitions.

Contributions of the paper. In this paper, we concentrate on the necessity of state predicate detection and state predicate correction within a pre-specified bounded amount of time. That is, we show that every fault-tolerant real-time program that reuses its fault-intolerant version must contain detectors or (weak/strong) δ-correctors. The main results from this paper are as follows:

- We precisely define what it means for a fault-tolerant real-time program to contain a component. Our definition of containment for (weak/strong) δ-correctors is similar to that in [1]. However, for detectors, our definition of containment is more rigorous than that in [1] in that it precisely identifies how predicates being detected are related to the *fault-intolerant* program and how the output of the detector (called witness predicate) is being used by the *fault-tolerant* program.
- A nonmasking fault-tolerant program with recovery time θ is one that recovers to a state from where its subsequent computation satisfies both (timed and untimed) safety and liveness within θ time units. However, safety may be violated before the program reaches such a state. We show that if a program satisfies both the safety and the liveness specifications within θ, then there exists δ, where δ is a function of θ, such that the program contains strong δ-correctors.
- A hard-failsafe fault-tolerant program is one that always satisfies both untimed and timed parts of its safety specification. Regarding hard-failsafe fault-tolerance, we show that hard-failsafe fault-tolerant programs contain detectors to satisfy the untimed part and weak δ-correctors to satisfy the timed part of their safety specification for some δ.
- A soft-masking fault-tolerant program with recovery time θ is one that recovers to a state from where its subsequent computation satisfies both safety and liveness within θ time units. Moreover, until such a state is reached, only the untimed part of its safety specification is not violated. We show that soft-masking fault-tolerant programs contain detectors and strong δ-correctors for some δ, where δ is a function of θ.
- A hard-masking fault-tolerant program with recovery time θ is a soft-masking program with recovery time θ that satisfies both untimed and timed parts of its safety specification during recovery. We show that hard-masking fault-tolerant programs contain detectors, weak δ_1-correctors, and strong δ_2-correctors for some δ_1 and δ_2, where δ_2 is a function of θ.

Organization of the paper. The rest of the paper is organized as follows: In Section 2, we formally define the notions of real-time programs, faults, and fault-tolerance. Then, in Section 3, we present the formal definition of fault-tolerance components for real-time programs, namely, detectors and weak/strong δ-correctors. In Section 4, we develop the theory of the components and we show the necessity of their existence in real-time programs with respect to different levels of fault-tolerance. Finally, in Section 5, we make concluding remarks and discuss future work.

2 Real-Time Programs, Specifications, and Fault-Tolerance

In this section, we give formal definitions of real-time programs, specifications, faults, and fault-tolerance. The definition of real programs is adapted from Alur and Henzinger [4] and Alur and Dill [5]. The definition of specifications is based on the work by Alpern and Schneider [2] and Henzinger [3]. Finally, the notion of faults and fault-tolerance is due to Arora and Gouda [6], and Bonakdarpour and Kulkarni [7].

2.1 Real-Time Programs

Let V be a finite set of *discrete variables* and W be a finite set of *clock variables*. Each discrete variable is associated with a finite *domain* D of values. A *location* is a function that maps each discrete variable to a value from its respective domain. A *clock constraint* over the set W of clock variables is a Boolean combination of formulas of the form $x \preceq c$ or $x - y \preceq c$, where $x, y \in W$, $c \in \mathbb{Z}_{\geq 0}$, and $\preceq$ is either $<$ or $\leq$. We denote the set of all clock constraints over W by $\Phi(W)$. A *clock valuation* is a function $\nu : W \rightarrow \mathbb{R}_{\geq 0}$ that assigns a real value to each clock variable. Furthermore, for $\tau \in \mathbb{R}_{\geq 0}$, $\nu + \tau = \nu(x) + \tau$ for every clock x. Also, for $\lambda \subseteq W$, $\nu[\lambda := 0]$ denotes the clock valuation for W which assigns 0 to each $x \in \lambda$ and agrees with ν over the rest of the clock variables in W.

A *state* (denoted σ) is a pair $\langle s, \nu \rangle$, such that s is a location and ν is a clock valuation for W at location s. The set of all possible states is called the *state space* obtained from the associated variables.

Definition 2.1 **(computations)** A *computation* is a finite or infinite timed state sequence of the form:

$$\overline{\sigma} = (\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots$$

where σ_i is a state in the state space obtained from the associated variables, for all $i \in \mathbb{Z}_{\geq 0}$, and the sequence $\tau_0\tau_1\tau_2 \cdots$ (called *global time*) satisfies the following constraints:

- *Initialization*: $\tau_0 \geq 0$,
- *Monotonicity*: $\tau_i \leq \tau_{i+1}$ for all $i \in \mathbb{Z}_{\geq 0}$, and
- *Divergence*: if $\overline{\sigma}$ is infinite, for all $t \in \mathbb{R}_{\geq 0}$, there exists j such that $\tau_j \geq t$. ∎

Notice that in Definition 2.1, we do not specify an initial value for global time. It follows that in a computation $\overline{\sigma}$, τ_0 can be assigned any value from $\mathbb{R}_{\geq 0}$. Let $\overline{\sigma} + t$ denote the computation with *time shift* $t \in \mathbb{R}$, such that $\tau_0 \geq 0$ (i.e., τ_i becomes $\tau_i + t$ for all $i \geq 0$). Also, let Σ be any set of computations. For all $\overline{\sigma} \in \Sigma$, we require that for all time shifts $t \in \mathbb{R}$, $\overline{\sigma} + t$ be in Σ as well.

Definition 2.2 **(suffix and fusion closure)** *Suffix closure* of a set of computations means that if a computation $\overline{\sigma}$ is in that set then so are all the suffixes of $\overline{\sigma}$. *Fusion closure* of a set of computations means that if computations $\langle \overline{\alpha}, (\sigma, \tau), \overline{\gamma} \rangle$ and $\langle \overline{\beta}, (\sigma, \tau), \overline{\psi} \rangle$ are in that set then so are the computations $\langle \overline{\alpha}, (\sigma, \tau), \overline{\psi} \rangle$ and $\langle \overline{\beta}, (\sigma, \tau), \overline{\gamma} \rangle$, where $\overline{\alpha}$ and $\overline{\beta}$ are finite prefixes of computations, $\overline{\gamma}$ and $\overline{\psi}$ are suffixes of computations, and σ is a state at global time τ. ∎

Notation. Let $\overline{\sigma}_i$ denote the pair (σ_i, τ_i) in computation $\overline{\sigma}$. Also, let $\overline{\alpha}$ and $\overline{\beta}$ be finite computations, where the length of $\overline{\alpha}$ is n. The concatenation of $\overline{\alpha}$ and $\overline{\beta}$ (denoted $\overline{\alpha}\overline{\beta}$) is a computation, where either clock variables (except possibly a subset that is reset) and global time of $\overline{\alpha}_{n-1}$ and $\overline{\beta}_0$ are equal, but their locations are different, or their locations are equal, but the clock variables and global time are advanced equally. If Γ and Ψ are two sets of finite computations, then $\Gamma\Psi = \{\overline{\alpha}\overline{\beta} : \overline{\alpha} \in \Gamma$ and $\overline{\beta} \in \Psi \}$.

Definition 2.3 **(real-time programs)** A *real-time program* p is specified by a set of discrete variables, a set of clock variables, and a suffix closed and fusion closed set of *maximal* computations in the state space of p (denoted S_p). By maximal, we mean that if $\overline{\sigma} = \overline{\alpha}\overline{\beta}$, where the prefix $\overline{\alpha} = (\sigma_0, \tau_0) \to (\sigma_1, \tau_1) \to \cdots (\sigma_n, \tau_n)$ and the infinite suffix $\overline{\beta} = (\langle s_{n+1}, \nu_{n+1}\rangle, \tau_{n+1}) \to (\langle s_{n+1}, \nu_{n+2}\rangle, \tau_{n+2}) \to (\langle s_{n+1}, \nu_{n+3}\rangle, \tau_{n+3}) \cdots$, is a computation of p such that $\nu_{j+1} = \nu_j + (\tau_{j+1} - \tau_j)$, for all $j > n$, then there is no other computation of p that has a prefix of $\overline{\alpha}$. In other words, given a finite computation prefix $\overline{\alpha}$ of p, p does not contain the computation that stutters σ_{n+1} infinitely if there is any other computation of p that extends $\overline{\alpha}$. ∎

Notation. For simplicity, we use the pseudo-arithmetic expressions to denote timing constraints over finite computations. For instance, $\overline{\sigma}_{\leq\delta}$, where $\delta \in \mathbb{Z}_{\geq 0}$, denotes a finite computation $(\sigma_0, \tau_0) \to (\sigma_1, \tau_1) \to \cdots (\sigma_n, \tau_n)$ that satisfies the timing constraint $\tau_n - \tau_0 \leq \delta$.

Definition 2.4 (state predicates) A *state predicate* of p is a subset of S_p. We say that a state predicate S is *closed* in p iff in every computation, $(\sigma_0, \tau_0) \to (\sigma_1, \tau_1) \to \cdots$, of p, if S is true in state σ_j, $j \in \mathbb{Z}_{\geq 0}$, (denoted $\sigma_j \models S$) then S remains true for all states σ_k, where $k \geq j$ (i.e., $\sigma_k \models S$). ∎

Definition 2.5 (S-computations) Let S be a state predicate and p be a program. The S-computations of p, denoted as $p \mid S$, is the set of computations of p that start in a state where S is true. ∎

Notice that since the set of computations of a program is suffix closed and fusion closed, the program can be written in terms of *transitions* that it can execute [1]. Hence, we can view a program p as a set of transitions. To concisely write the transitions of a program, we use *timed guarded commands* [4]. A timed guarded command (also called *timed actions*) is of the form $L :: \mathit{Guard} \xrightarrow{\lambda} \mathit{statement}$, where L is a label, *Guard* is a state predicate, $\mathit{statement}$ is a statement that describes how the program state is updated, and λ is a set of clock variables that are reset by execution of L. Thus, L denotes the set of transitions $\{(\sigma_0, \sigma_1) \mid \mathit{Guard}$ is true in state σ_0, σ_1 is obtained by resetting the clock variables in λ and changing σ_0 as prescribed by $\mathit{statement}\}$. A *guarded wait command* (also called *delay action*) is of the form $\mathit{Guard} \longrightarrow \textbf{wait}$, where Guard identifies the states where a delay transition is allowed to be taken (i.e., the program stutters in a location and lets time advance). A guarded wait command delays the program by an arbitrary amount of time, as long as Guard remains true. We present examples of timed guarded commands in Section 4.

2.2 Specifications

Similar to a program, a *specification* (also called *property*) is specified by sets of discrete and clock variables (respectively, state space) and a suffix closed and fusion closed set of (finite or infinite) computations over the state space obtained from those variables.

In order to capture the real-time properties of programs (e.g., deadlines and recovery time), in this paper, we focus on a standard property of real-time systems called *stable bounded response*.

Definition 2.6 (stable bounded response) Let P and Q be two state predicates. A *stable bounded response* property (denoted $P \mapsto_{\leq\delta} Q$) is the set of all computations $(\sigma_0, \tau_0) \to (\sigma_1, \tau_1) \to \cdots$ in which if $\sigma_i \models P$, for some $i \geq 0$, then there exists j, $j \geq i$, such that (1) $\sigma_j \models Q$, (2) $\tau_j - \tau_i \leq \delta$, and (3) for all k, $i \leq k < j$, $\sigma_k \models P$. In other words, it is always the case that a state in P is followed by a state in Q within δ time units and P continuously remains true until Q becomes true. We call P the *event predicate*, Q the *response predicate*, and δ the *response time*. ∎

Assumption 2.7 We assume that the set of clock variables of any stable bounded response property $P \mapsto_{\leq \delta} Q$ contains a special clock variable, which is reset whenever P becomes true. This assumption is necessary to ensure that stable bounded response properties are fusion closed. ∎

The specifications considered in this paper (denoted $SPEC$) are an intersection of the safety specification and the liveness specification, defined next.

Definition 2.8 (safety specification) We define the *safety specification* by a set of computations based on (1) a set $SPEC_{bt}$ of instantaneous *bad transitions* of the form $\langle s_0, \nu \rangle \rightarrow \langle s_1, \nu[\lambda := 0] \rangle$ where s_0 and s_1 are two locations and λ is a set of clock variables, and (2) a set $SPEC_{br}$ of m bounded response properties of the form $(P_1 \mapsto_{\leq \delta_1} Q_1) \wedge (P_2 \mapsto_{\leq \delta_2} Q_2) \wedge \ldots \wedge (P_m \mapsto_{\leq \delta_m} Q_m)$, for some m, $m \geq 1$. Precisely, the safety specification is the intersection of the following sets:

1. the set of computations where no prefix contains a transition in $SPEC_{bt}$, and
2. the intersection of sets of computations corresponding to each stable bounded response property $P_i \mapsto_{\leq \delta_i} Q_i$ in $SPEC_{br}$, where $1 \leq i \leq m$. ∎

Notation. With abuse of notation for simplicity, throughout the paper, whenever we refer to $SPEC_{bt}$, we mean the corresponding set of computations that do not contain a transition in $SPEC_{bt}$.

Definition 2.9 (liveness specification) A liveness specification is a set of computations that meets the following condition: for each finite computation $\overline{\alpha}$ there exists a computation $\overline{\beta}$ such that $\overline{\alpha}\overline{\beta}$ is in that set. ∎

Notation. We use S^* to denote a finite computation $(\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots (\sigma_n, \tau_n)$ such that $\sigma_i \models S$ for all i, where $0 \leq i \leq n$. Thus, $(true)^*$ denotes an arbitrary finite computation.

Now, we define what it means for program p to refine a specification $SPEC$, and what it means for program p' (typically, a fault-tolerant program) to refine program p (typically, a fault-intolerant program). Essentially, we would like to define 'p' refines p' iff computations of p' are a subset of that in p. However, if p' is obtained by adding fault-tolerance to p then p' may contain additional variables that are not in p. Hence, it will be necessary to project the computations of p' on (the variables of) p and then check if the projected computation is a computation of p. More precisely, the *projection* of a state of p' on p (respectively, $SPEC$) is a state obtained by considering only the (discrete and clock) variables of p (respectively, $SPEC$). Extending this definition for computations, we say that the projection of a computation of p' on p (respectively, $SPEC$) is a computation obtained by projecting each state in that computation on p (respectively, $SPEC$).

Definition 2.10 (refines) We say that p' *refines* p (respectively, $SPEC$) *from* S iff the following two conditions hold: (1) S is closed in p', and (2) For every computation of p' that starts in a state where S is true, the projection of that computation on p (respectively, $SPEC$) is a computation of p (respectively, $SPEC$). ∎

We also define the notion of *maintains* for finite computations. Specifically, given a finite prefix $\overline{\alpha}$, $\overline{\alpha}$ maintains $SPEC$ captures that the specification is not yet violated in $\overline{\alpha}$.

Definition 2.11 (maintains) Let $\overline{\alpha}$ be a prefix of a computation of p. We say that $\overline{\alpha}$ *maintains* $SPEC$ iff there exists a computation $\overline{\beta}$ such that the projection of $\overline{\alpha}\overline{\beta}$ on $SPEC$ is in $SPEC$. ∎

Informally speaking, proving the correctness of p with respect to $SPEC$ involves showing that p refines $SPEC$ from some state predicate S, where $S \neq \{\}$. We call such a state predicate S an invariant of p.

Definition 2.12 (invariant) Let S be a state predicate. We say that S is an *invariant of* p *for* $SPEC$ iff p refines $SPEC$ from S and $S \neq \{\}$. ∎

2.3 Faults and Fault-Tolerance in Real-Time Programs

Intuitively, the faults that a program is subject to are systematically represented by the union of transitions whose execution perturbs the program state and transitions that unexpectedly advance time. While state corruption faults may indirectly cause waste of time, delay faults directly cause waste of time in the sense that they defer the occurrence of some desirable event by some amount of time. For instance, a processor crash may require a scheduler to assign another processor to a set of jobs. It is natural, to model the delay in start time of such jobs by delay faults that only advance the value of clock variables. Formally, we model faults as a set of transitions over (discrete and clock) variables of p.

Definition 2.13 (faults) For a program p with state space S_p, the set f of *faults* is specified by the union of the following two sets:

1. the set f^s of *immediate faults* of the from $\langle s_0, \nu\rangle \rightarrow \langle s_1, \nu[\lambda := 0]\rangle$ where s_0 and s_1 are two locations and λ is a set of (possibly empty) clock variables, and
2. a set f^t of *delay faults* of the form $\langle s, \nu\rangle \rightarrow \langle s, \nu + \tau\rangle$, which keeps the program in a location for some time $\tau \in \mathbb{R}_{\geq 0}$.

We now define what we mean by computations of a program in the presence of faults. Given a program p and faults f, we define the computations of p in the presence of f by finite fusion-closure of the computations of $p \cup f$ as follows. Let Z be a set of computations and $\bullet$ be the operator that fuses two (finite or infinite) computations of Z such that $\bullet(Z) = \{\overline{\alpha}(\sigma,\tau)\overline{\beta} \mid \exists \overline{\gamma}, \overline{\psi} \ : \ (\overline{\alpha}(\sigma,\tau)\overline{\gamma} \in Z) \ \wedge \ (\overline{\psi}(\sigma,\tau)\overline{\beta} \in Z)\}$. Also, let $FFC(Z)$ be the smallest fixpoint of $\cup_{i=0}^{\infty} \bullet^i (Z)$. Now, we define the *computations of* p *in the presence of* f (denoted $p[]f$) as $FFC(p \cup f)$.

Assumption 2.14 Observe that the above formulation of program computations in the presence of faults guarantees that the number of occurrence of faults in a computation is finite. In this paper, however, since we deal with real-time programs and our goal is to identify components that provide "bounded-time" recovery in the presence of faults, we assume that the number of occurrence of faults in all computations is bounded by some number $k \in \mathbb{Z}_{\geq 0}$. This assumption is reasonable in many commonly considered fault-tolerant real-time programs. In fact, it can be shown that providing bounded-time recovery in the presence of unbounded number of faults is not possible. ∎

Just as we use invariants to show program correctness in the absence of faults, we use *fault-spans* to show the correctness of programs in the presence of faults.

Definition 2.15 **(fault-span)** Let S and T be state predicates and f be a set of fault transitions. We say that T is an *f-span of p from S* iff $S \subseteq T$, and T is closed in $p[]f$. ∎

Notation. Henceforth, if the specification or projection of a program on $SPEC$ is clear from the context, we omit it. For example, "S is an invariant of p" abbreviates "S is an invariant of p for $SPEC$". Likewise, "a computation of p is in $SPEC$" abbreviates "the projection of that computation on $SPEC$ is in $SPEC$". Similarly, "T is an f-span of p" abbreviates "T is an f-span of p from S".

We now define what we mean by levels of fault-tolerance in the context of real-time programs. Obviously, in the absence of faults, a program should refine its specification. In the presence of faults, however, it may not refine its specification and, hence, it may refine some (possibly) weaker 'tolerance specification'. These specifications are based on satisfaction of a combinations of safety, liveness, timing constraints, and a desirable bounded-time recovery mechanism in the presence of faults. Intuitively, we consider three levels of fault-tolerance, namely *failsafe, nonmasking, and masking*, based on satisfaction of safety and liveness properties in the presence of faults. For failsafe and masking fault-tolerance, we, furthermore, consider two additional levels, namely *soft* and *hard*, based on satisfaction of timing constraints in the presence of faults. Moreover, nonmasking and masking fault-tolerant programs are associated with a required *recovery time*, which can be *bounded* or *unbounded*.

To motivate the idea of soft and hard fault-tolerance, let us consider the railroad crossing problem. Suppose that a train is approaching a railroad crossing. The safety specification requires that "if the train is crossing, the gate must be closed". Also, a stable bounded response property requires that "once the gate is closed, it should reopen within 5 minutes". In this example, it may be catastrophic if the train is crossing while the gate is open due to occurrence of faults. On the other hand, if the gate remains closed for more than 5 minutes due to occurrence of faults, the outcome is not disastrous. Thus, depending upon the outcome of violation of a safety specification, the desired level of fault-tolerance changes. Hence, in the railroad crossing problem the system must tolerate faults that cause the gate to remain open while the train is crossing. We call such a system *soft fault-tolerant*. Intuitively, a soft fault-tolerant real-time program is not required to satisfy its timing constraints in the presence of faults.

Now, consider a system that controls internal pressure of a boiler. Suppose that in this system, the safety specification requires that once a pressure gauge reads 30 pounds per square inch, the controller must issue a command to open a valve within 20 seconds. In such a system, if occurrence of faults causes the controller not to respond within the required time, the outcome may be disastrous. Thus, our boiler controller must satisfy its timing constraints even in the presence of faults. In other words, the boiler controller must be *hard fault-tolerant*. Intuitively, a hard fault-tolerant real-time program must satisfy its timing constraints even in the presence of faults. In fact, in hard fault-tolerant programs, the demand for hard real-time processing merges with catastrophic consequences of systems, whereas in soft fault-tolerance the catastrophic consequences are not related to the program's timing constraints.

Below, we define tolerance specifications that often occur in practice. Let $SPEC$ be the specification obtained by the intersection of $SPEC_{bt}$, $SPEC_{br}$, and liveness specification as defined in Subsection 2.2.

Definition 2.16 **(soft and hard-failsafe tolerance specification)** The *soft-failsafe tolerance specification* of $SPEC$ is the smallest safety specification containing $SPEC_{bt}$ (denoted $SSPEC_{bt}$). The *hard-failsafe tolerance specification* of $SPEC$ is the intersection of $SSPEC_{bt}$ and the smallest specification containing $SPEC_{br}$ (denoted $SSPEC_{br}$). I.e, The hard-failsafe tolerance specification of $SPEC$ is $SSPEC_{bt} \cap SSPEC_{br}$. ∎

Definition 2.17 (nonmasking tolerance specification) Since in nonmasking fault-tolerance it is not required to satisfy the safety specification in the presence of faults, the *nonmasking tolerance specification of $SPEC$ with recovery time θ* is $(true)^*_{\leq\theta} SPEC$. ∎

Definition 2.18 (soft and hard-masking tolerance specification) The *soft-masking tolerance specification of $SPEC$ with recovery time θ* is $SPEC_{bt \leq \theta} SPEC$. The *hard-masking tolerance specification of $SPEC$ with recovery time θ* is $SPEC$.

Remark. Notice that the hard-masking tolerance specification is independent of the recovery time θ. This is because unlike nonmasking and soft-masking, in hard-masking fault-tolerance the entire specification (i.e., $SPEC$) must always be satisfied and, hence, recovery time becomes only a matter of the amount of time that the program is in states outside its invariant.

Using these definitions, we are now ready to define what it means for a program to tolerate a fault-class f. With the intuition that a program is f-tolerant to $SPEC$ if it refines $SPEC$ in the absence of faults and it refines a tolerance specification of $SPEC$ in the presence of f, we define 'f-tolerant to $SPEC$ from S' as follows.

Definition 2.19 (fault-tolerant programs) We say that p is *soft/hard-failsafe f-tolerant to $SPEC$ from S* (respectively, nonmasking or soft/hard-masking f-tolerant to $SPEC$ with recovery time θ from S) iff the following two conditions hold:

- p refines $SPEC$ from S, and
- there exists T such that $T \supseteq S$ and $p[]f$ refines the soft/hard-failsafe tolerance specification of $SPEC$ from T (respectively, the nonmasking or soft/hard-masking tolerance specification of $SPEC$ with recovery time θ from T). ∎

Note that for nonmasking and masking levels of fault-tolerance one can choose to have unbounded-time recovery. We address the effect of the choice of recovery time in Section 4.

Notation. In the sequel, whenever the specification $SPEC$ and the invariant S are clear from the context, we omit them; thus, "nonmasking f-tolerant with recovery time θ" abbreviates "nonmasking f-tolerant to $SPEC$ with recovery time θ from S", and so on.

3 Real-Time Fault-Tolerance Components

In this section, we present real-time fault-tolerance components, namely, *detectors*, *weak δ-correctors*, and *strong δ-correctors*. Once we define these components, in Section 4, we present the relevance of each component to the levels of fault-tolerance introduced in Subsection 2.3.

3.1 Detectors

In this subsection, we formally introduce the first of the three tolerance components, *detectors*. We will develop their theory and present a simple altitude switch example to illustrate an instance of detectors in Subsection 4.2.

Definition 3.1 **(detects)** Let X and Z be state predicates. Let 'Z *detects* X' be the specification, that is the set of all computations $\overline{\sigma} = (\sigma_0, \tau_0) \to (\sigma_1, \tau_1) \to \cdots$, satisfying the following three conditions:

- (*Safeness*) For each $i, i \geq 0$, if $\sigma_i \models Z$ then $\sigma_i \models X$. (In other words, $\sigma_i \models (Z \Rightarrow X)$.)
- (*Progress*) For each $i, i \geq 0$, if $\sigma_i \models X$ then there exists $k, k \geq i$, such that $\sigma_k \models Z$ or $\sigma_k \not\models X$.
- (*Stability*) There exists $i, i \geq 0$, such that for all $j, j \geq i$, if $\sigma_j \models Z$ then $\sigma_{j+1} \models Z$ or $\sigma_{j+1} \not\models X$. ∎

Definition 3.2 **(detector)** Z *detects* X *in* d from U iff d refines 'Z detects X' from U. ∎

A detector d is used to check whether its "detection predicate", X, is true. Since d satisfies *Progress* from U, in any computation in d, if $U \wedge X$ is true continuously, d eventually detects this fact and makes Z true. Since d satisfies *Safeness* from U, it follows that d never lets Z witness X incorrectly. Moreover, since d satisfies *Stability* from U, it follows that once Z becomes true, it continues to be true unless X is falsified.

Definition 3.3 **(tolerant detector)** d is a *soft/hard-failsafe* (respectively, nonmasking or soft/hard-masking) *tolerant detector* (respectively, with recovery time θ) to 'Z detects X' from U iff d refines the soft/hard-failsafe (respectively, nonmasking or soft/hard-masking) tolerance specification of 'Z detects X' (respectively, with recovery time θ) from U. ∎

3.2 δ-Correctors

In this subsection, we formally introduce the second of the two tolerance components, *δ-correctors*. We will develop their theory and present examples to illustrate an instance of δ-correctors in subsections 4.1, 4.2, and 4.3.

Definition 3.4 **(weakly corrects)** Let X and Z be state predicates. Let 'Z *weakly corrects* X *within* δ' be the specification, that is the set of all computations $\overline{\sigma} = (\sigma_0, \tau_0) \to (\sigma_1, \tau_1) \to \cdots$, satisfying the following conditions:

- (*Week Convergence*) There exists $i, i \geq 0$, such that $\sigma_i \models X$ and $(\tau_i - \tau_0) \leq \delta$.
- (*Safeness*) For each $i, i \geq 0$, if $\sigma_i \models Z$ then $\sigma_i \models X$. (In other words, $\sigma_i \models (Z \Rightarrow X)$.)
- (*Progress*) For each $i, i \geq 0$, if $\sigma_i \models X$ then there exists $k, k \geq i$, such that $\sigma_k \models Z$ or $\sigma_k \not\models X$.
- (*Stability*) There exists $i, i \geq 0$, such that for all $j, j \geq i$, if $\sigma_j \models Z$ then $\sigma_{j+1} \models Z$ or $\sigma_{j+1} \not\models X$. ∎

Definition 3.5 **(weak δ-corrector)** Z *weekly corrects* X *within* δ *in* c from U iff c refines 'Z weakly corrects X within δ' from U. ∎

Definition 3.6 **(strongly corrects)** Let X and Z be state predicates. Let 'Z *strongly corrects* X *within* δ' be the specification, that is the set of all computations $\overline{\sigma} = (\sigma_0, \tau_0) \to (\sigma_1, \tau_1) \to \cdots$, satisfying the following two conditions:

- Z weakly corrects X within δ, and
- (*Strong Convergence*) In addition to Weak Convergence, X is closed in $\overline{\sigma}$. ∎

Definition 3.7 **(strong δ-corrector)** We say that Z *strongly corrects* X *within* δ *in* c from U iff c refines 'Z strongly corrects X within δ' from U. ∎

Since c satisfies Weak (respectively, Strong) Convergence from U, it follows that c reaches a state where X becomes true within δ time units (and, respectively, X continues to be true thereafter). Moreover, since c satisfies Safeness from U, it follows that a corrector never lets the predicate Z witness the correction predicate X incorrectly. Since c satisfies Progress from U, it follows that eventually Z becomes true. And, finally, since c satisfies Stability from U, it follows that when Z becomes true, Z is never falsified.

Definition 3.8 **(tolerant δ-corrector)** c is a *soft/hard-failsafe* (respectively, nonmasking or soft/hard-masking) *tolerant weak/strong δ-corrector* (respectively, with recovery time θ) to 'Z weakly/strongly corrects X within δ' from U iff c refines the soft/hard-failsafe (respectively, nonmasking or soft/hard-masking) tolerance specification (respectively, with recovery time θ) of 'Z weakly/strongly corrects X within δ' from U. ∎

Remark. If the witness predicate Z is identical to the correction predicate X, our definition of the corrects relation reduces to one given by Arora and Gouda [6]. We have considered this more general definition to accommodate the case —which occurs for instance in soft/hard-masking tolerance— where the witness predicate Z can be checked atomically but the correction predicate X cannot.

4 The Necessity of Detectors and δ-Correctors in Fault-Tolerant Real-Time Programs

In this section, we develop the theory of detectors and δ-correctors. More specifically, we show the necessity of existence of a combination of these components in different levels of fault-tolerance introduced in Subsection 2.3.

Observe that, in the definition of tolerance specifications in Subsection 2.3, when the required recovery time for nonmasking and soft-masking fault-tolerance is unbounded, the resulting levels of fault-tolerance reduce to the ones addressed by Arora and Kulkarni in [1]. Moreover, since soft-failsafe programs are not required to satisfy their timing constraints in the presence of faults, this level reduces to the one considered in [1] as well. Thus, in the following subsections, out of eight possible levels of fault-tolerance introduced in Subsection 2.3, we only consider the ones that are not addressed in [1].

4.1 The Role of Strong δ-Correctors in Nonmasking Fault-Tolerance

In this subsection, we start developing the theory of strong δ-correctors. Subsequently, we build upon an example to illustrate an instance of δ-correctors. Intuitively, our goal is to show that if a program is nonmasking f-tolerant then it contains nonmasking tolerant strong δ-correctors. We present this result in Theorem 4.4. Throughout this section, let p be a program, $SPEC$ be a specification, and σ be a state.

In order to prove our goal, first, in Claim 4.2, we show that if a program refines a specification within θ time units then it contains strong δ-correctors for some δ. More specifically, let p be a program that refines $SPEC$ from S. In Claim 4.2, we show that if p' is designed such that it behaves like p within θ and, thus, has a suffix in $SPEC$, then p' is a strong δ-corrector of an invariant predicate of p for some δ. We prove this Claim by showing that p' itself refines the required strong δ-corrector specification.

Definition 4.1 **(becomes)** We say that p' *becomes* p *within* θ *from* T iff p' refines $(true)^*_{\leq\theta}p$ from T. ∎

Claim 4.2 *Programs that refine a specification within θ time units contain strong δ-correctors. That is,*

if

- *p refines SPEC from S,*
- *p' refines p from S, and*
- *p' becomes $p|S$ within θ from T*

then

- *there exists $\delta \in \mathbb{Z}_{\geq 0}$ such that p' is a strong δ-corrector of an invariant predicate of p.*

Proof.

Let $X = S$, and

$Z = S \wedge \{\sigma \mid \sigma$ is a state of p' such that σ is reached in some computation of p' starting from $T\}$.

Since p refines $SPEC$ from S, it follows that X is an invariant predicate of p for $SPEC$. Now, we show that there exists $\delta \in \mathbb{Z}_{\geq 0}$ such that p' refines 'Z strongly corrects X within δ' from T:

- By definition of Z, in any state where Z is true, S is true. Thus, Safeness is satisfied.
- Since p' becomes $p|S$ within θ from T, i.e., p' refines $(true)^*_{\leq\theta}(p|S)$ from T, every computation of p' starting from T will reach a state where S is true within θ. By definition of Z, Z is true in this state as well. Thus, Progress is satisfied.
- Since p' refines p from S, it follows that S is closed in p'. Obviously, the second conjunct in Z is closed in p' as well and, thus, Z is closed in p'. Moreover, since we are guaranteed that Z is eventually reached in any computation, Stability is satisfied.
- Since p' refines $(true)^*_{\leq\theta}(p|S)$ from T, every computation of p' starting from T will reach a state where S is true within θ. Moreover, $S(= X)$ is closed in p'. Thus, by choosing $\delta = \theta$, Strong Convergence within δ is satisfied. ∎

The next lemma generalizes Claim 4.2. In general, given a program p that refines $SPEC$ from S, p' may not behave like p from each state in S but only from a subset of S, say R. This may happen, for example, if p' contains additional variables and p' behaves like p only after the values of these additional variables are restored. Lemma 4.3 shows that in such a case, p' contains a nonmasking tolerant strong δ-corrector of an invariant predicate of p. (The strong δ-corrector is nonmasking in the sense that the correction predicate is preserved only after p' reaches a state where R is true.)

Lemma 4.3

if

- *p refines SPEC from S,*

- *p' refines p from R, where $R \Rightarrow S$, and*
- *p' becomes $(p|R)$ within θ from T*

then

- *there exists $\delta \in \mathbb{Z}_{\geq 0}$ such that p' is a nonmasking tolerant strong δ-corrector with recovery time θ of an invariant predicate of p.*

Proof.

Let $X = S$, and

$Z = R$

We show that there exists $\delta \in \mathbb{Z}_{\geq 0}$ such that p' refines the nonmasking tolerance specification with recovery time θ of 'Z strongly corrects X within δ' from T. In particular, we first show that a computation of p' starting from a state where T is true reaches a state where R is true within θ time units. Then, we show that starting from this state p' refines 'Z strongly corrects X within δ'.

- For the first part, since p' becomes $p|R$ within θ from T, it follows that p' reaches a state where R is true within θ.
- For the second part, we show that there exists $\delta \in \mathbb{Z}_{\geq 0}$ such that starting from this state, p' satisfies Safeness, Progress, Stability and Strong Convergence within δ. Since $R \Rightarrow S$ is trivially true, Safeness is satisfied. In a state where R is true, Progress is satisfied. Since p' refines p from R and R is closed in p', Stability is satisfied. Finally, since in a computation starting from a state where R is true, S is immediately true at all states and p' is closed in S, by choosing $\delta = 0$ Strong Convergence is satisfied. ∎

We now use Claim 4.2 and Lemma 4.3 to show that if a nonmasking f-tolerant program p' with recovery time θ is designed by reusing p then there exists $\delta \in \mathbb{Z}_{\geq 0}$ such that p' contains a nonmasking f-tolerant strong δ-corrector with recovery time θ for an invariant predicate of p.

Theorem 4.4 *Nonmasking f-tolerant programs with recovery time θ contain nonmasking tolerant strong δ-correctors with recovery time θ', for some θ'. That is,*
if

- *p refines SPEC from S,*
- *p' is nonmasking f-tolerant for SPEC from R, where $R \Rightarrow S$,*
- *p' refines p from R, and*
- *p' becomes $p|R$ within θ from T where T is an f-span of p'*

then

- *there exists δ and θ' in $\mathbb{Z}_{\geq 0}$ such that p' is a nonmasking f-tolerant strong δ-corrector with recovery time θ' of an invariant predicate of p.*

Proof. Let T_1 be the fault-span used to show that p' is nonmasking fault-tolerant. Let $T_2 = T_1 \wedge T$. Since T_1 and T are

f-spans, both are closed in $p'[]f$. Hence T_2 is closed in $p'[]f$. Moreover, since $T_2 \subseteq T$, p' becomes $p|R$ from T_2.

Now, we use the definition of Z and X given in the proof of Lemma 4.3 and show that there exists $\delta, \theta \in \mathbb{Z}_{\geq 0}$ such that p' is nonmasking f-tolerant with recovery time θ' to 'Z strongly corrects X within δ' from R and the f-span of p' is T_2. To this end, we first show that there exists $\delta \in \mathbb{Z}_{\geq 0}$ such that p' refines 'Z strongly corrects X within δ' from R, and then show that $p'[]f$ refines the nonmasking tolerance specification with recovery time θ of 'Z strongly corrects X within δ' from T_2.

In Lemma 4.3, we have shown that there exists $\delta \in \mathbb{Z}_{\geq 0}$ such that starting from any state in R, every computation of p' satisfies Safeness, Progress, Stability, and Strong Convergence within δ. It follows that by choosing $\delta = 0$, p' refines 'Z strongly corrects X within δ' from R.

The proof of Lemma 4.3 can also be used to show that there exists $\delta \in \mathbb{Z}_{\geq 0}$ such that p' refines the nonmasking tolerance specification with recovery tine θ of 'Z strongly corrects X within δ' from T_2. In the presence of f, this specification may be violated. However, by Assumption 2.14, after at most k faults occur, where k is the bound on the number faults that may occur in a computation, p' reaches a state where R is true within at most $k.\theta$ time units (i.e., $\theta' = k.\theta$). And, from this state, p' refines 'Z strongly corrects X within $\delta(= 0)$'. Thus, p' is nonmasking f-tolerant with recovery time $\theta' = |f^s|.\theta + \sum f^t$ to 'Z strongly corrects X within $\delta(= 0)$' from R. ∎

In the proof of Theorem 4.4, although we showed that the recovery time that a strong δ-corrector provides in worst case depends on the maximum number of occurrence of faults, one can design correctors with better recovery time. For instance, in [7], the authors present automated methods to synthesize one type of such correctors using state exploration and Dijkstra's shortest path algorithm.

Remark. As mentioned earlier in this section, if we do not require bounded-time recovery for nonmasking programs, the results presented in this subsection reduce to the one given by Arora and Kulkarni [1], in which nonmasking tolerant correctors are defined as strong ∞-correctors.

4.1.1 Example

Consider an *alternating bit protocol* where an infinite input array at a sender process is to be copied, one array item at a time, into an infinite output array at a receiver process. The sender and receiver communicate via a bidirectional channel that can hold at most one message in each direction at a time.

Fault-intolerant program. Iteratively, a simple loop is followed: sender s sends a copy of one array item to receiver r. Upon receiving this item, r sends an acknowledgment to s, which enables the next array item to be sent by s and so on. Each iteration is supposed to take at most τ time units. The program maintains binary variables rs in s and rr in r; rs is 1 if s has received and acknowledgement for the last message it sent, and rr is 1 if r has received a message but has not sent an acknowledgement yet. Furthermore, the program maintains a timer x which is reset every time s sends a message. Also, s receives an acknowledgement if the timer is not expired. The 0 or 1 messages in transit from s to r are denoted by the sequence cs, and the 0 or 1 acknowledgments in transit from r to s are denoted by the sequence cr. Finally, the index in the input array corresponding to the item that s will send next is denoted by ns, and the index in the output array corresponding to the item that r last received is denoted by nr.

The intolerant program contains five actions, the first two in s and the next two in r. The last action is a guarded wait command, which models legitimate delays. The specification $SPEC$ requires that each item is delivered to the receiver and the sender receives the corresponding acknowledgement within 3 time units.

$ID1 :: rs = 1 \quad \xrightarrow{\{x\}} \quad rs, cs := 0, cs \circ \langle ns \rangle$

$ID2 :: cr \neq \langle\rangle \wedge x \leq \tau \quad \longrightarrow \quad rs, cr, ns := 1, tail(cr), ns + 1$

$ID3 :: cs \neq \langle\rangle \quad \longrightarrow \quad cs, rr, nr := tail(cs), 1, head(cs)$

$ID4 :: rr = 1 \quad \longrightarrow \quad rr, cr := 0, cr \circ \langle nr \rangle$

$ID5 :: x \leq 3 \quad \longrightarrow \quad$ **wait**

Invariant. When r receives a message, $nr = ns$ holds, and this equation continues to hold until s receives and acknowledgement. When s receives an acknowledgement, ns is exactly one greater than nr and this equation continues to hold until r receives the next message. Also, if cs is nonempty, cs contains only one item, $\langle ns \rangle$. Finally, at any state exactly two actions are enabled: one of $ID1 ... ID4$ and $ID5$. Thus the invariant of the fault-intolerant program is:

$$S_{ID} = (x \leq \tau) \ \wedge \ ((rr = 1 \ \vee \ cr \neq \langle\rangle)) \ \wedge \ ((rs = 1 \ \vee \ cs \neq \langle\rangle) \ \Rightarrow \ nr = ns - 1) \ \wedge$$
$$(cs = \langle\rangle \ \vee \ cs = \langle ns \rangle) \ \wedge \ (|cs| + |cr| + rs + rr = 1)$$

Fault Actions. In this example, the immediate fault actions cause loss of either a message or an acknowledgment. The delay faults expire the transmission timer and may advance it for at most 2 time units. The corresponding fault actions are as follows:

$cs \neq \langle\rangle \quad \longrightarrow \quad cs := tail(cs)$

$cr \neq \langle\rangle \quad \longrightarrow \quad cr := tail(cr)$

$3 \leq x \leq 5 \quad \longrightarrow \quad$ **wait**

Nonmasking fault-tolerant program. In order to add nonmasking fault-tolerance with recovery time $\theta = 10$, we design and add a strong δ-corrector CR with correction predicate $X = S_{ID}$ (i.e., the invariant of the intolerant program), witness predicate $Z = S_{ID} \ \wedge \ \{\sigma \mid x(\sigma) = 0\}$ (i.e., states of the invariant where x is reset), and $\delta = 13$. Precisely, CR consists of two actions: (1) an action that retransmits the last sent item, and (2) a delay action that maintains the required recovery time θ. Thus, the actions of the corrector CR are as follows:

$CR1 :: (cs = \langle\rangle \ \wedge \ cr = \langle\rangle \ \wedge \ rs = 0 \ \wedge \ rr = 0) \vee (x > \tau) \quad \xrightarrow{\{x\}} \quad cs := cs \circ \langle ns \rangle$

$CR2 :: 3 \leq x \leq 13 \quad \longrightarrow \quad$ **wait**

Thus, the nonmasking program consists of seven actions; five actions are identical to the actions of program ID, the rest are the actions of the corrector CR identified above.

4.2 The Role of Detectors and Week δ-Correctors in Hard-Failsafe Fault-Tolerance

In this Subsection, we present a more rigorous theory of detectors than the one presented in [1] and develop the theory weak δ-correctors. Intuitively, in this subsection, our goal is to show that if a program is hard-failsafe f-tolerant then it contains hard-failsafe tolerant detectors and weak δ-correctors. We present this result in Theorem 4.13.

Our proof is organized as follows: first, Lemma 4.6 shows that the violation of the untimed part of the safety specification can be detected from the current state, independent of how that state is reached. Subsequently, Lemma 4.7 shows that there exists a set of states from where execution of the program maintains the given safety specification. Next, using lemmas 4.6 and 4.7, in Lemma 4.9, we show that for any program p, there exists a unique weakest detection predicate. Then, in Claim 4.12, we prove that if a program refines a safety specification then it contains detectors and weak δ-correctors. Finally, in Theorem 4.13, we show that hard-failsafe programs contain hard-failsafe tolerant detectors and weak δ-correctors.

Throughout this section, let p be a program, $SPEC$ be a specification, $SSPEC_{bt}$ and $SSPEC_{br}$ be the minimal safety specifications that contains $SPEC_{bt}$ and $SPEC_{br}$ respectively, $\overline{\alpha}$ be a prefix of a computation, $\overline{\beta}$ be a finite suffix of a computation, σ and σ' be states, and X be a state predicate.

Assumption 4.5 Without loss of generality, for simplicity, we assume that transitions that correspond to different actions of the program (except guarded wait commands) are mutually disjoint, i.e., they do not contain overlapping transitions. The results int this paper are valid without this assumption in the sense that we can easily modify the given program to one that satisfies this assumption. Moreover, we assume that program computations are *fair* in the sense that in every computation, if the guard of an action is continuously true then that action is eventually chosen for execution.

Recall that in order to show the existence of the correctors in Subsection 4.1, we proved that the fault-tolerant program contains a corrector for an invariant predicate of the fault-intolerant program. For existence of detectors, we would like to show that the fault-tolerant program contains a detector for a detection predicate associated with the fault-intolerant program. However, unlike the invariant predicate, we need to identify additional syntactic characteristics of program before detection predicates can be identified. With this intuition, we now characterize programs syntactically. First, we reiterate two lemmas from [1]. Note that in Lemma 4.6, since the transitions in $SPEC_{bt}$ occur instantaneously, we ignore the global time of states.

Lemma 4.6 (Arora and Kulkarni [1])

If

- $\overline{\alpha}\sigma$ *maintains* $SPEC_{bt}$

then

- $\overline{\alpha}\sigma\sigma'$ *maintains* $SPEC_{bt}$ *iff* $\sigma\sigma'$ *maintains* $SPEC_{bt}$. ∎

Lemma 4.7 (Arora and Kulkarni [1]) *Given program p, there exists a predicate such that execution of p in a state where that predicate is true maintains $SPEC_{bt}$.* ∎

Definition 4.8 (detection predicate) We say that X is a *detection predicate* of p for $SPEC_{bt}$ iff execution of p in any state where X is true maintains $SPEC_{bt}$. ∎

We now prove the uniqueness of the weakest detection predicate for a given program p.

Lemma 4.9 *Given a program p, there exists a unique weakest detection predicate of p for $SPEC_{bt}$.*

Proof. Note that the existence of detection predicates follows from Lemmas 4.6 and 4.7, and that a program may have multiple detection predicates. Now, let sf be a detection predicate of p for $SPEC_{bt}$ and X be an arbitrary state predicate. It is easy to see that if $X \Rightarrow sf$, then X is also a detection predicate of p for $SPEC_{bt}$. And, if sf_1 and sf_2 are detection predicates of p for $SPEC_{bt}$ then so is $sf_1 \vee sf_2$. Thus, there exists a unique weakest detection predicate for the given program. ∎

Constraints for demonstrating existence of detectors. Now, using the notion of timed guarded commands (cf. Section 2), we precisely define what it means for a fault-tolerant program to contain detectors. In particular, given a timed guarded command, say $g \xrightarrow{\lambda} st$, Lemma 4.9 shows that there is a unique *weakest* safe predicate, say sf, for it. Hence, to show the existence of detectors, we require that the detection predicate of such an action is $g \wedge sf$. Furthermore, to show that the fault-tolerant program *contains* the desired detector, we show that it must be using the witness predicate of that detector to ensure that execution of the corresponding action is safe. Towards this end, we define the notion of *encapsulation*. Intuitively, if p' encapsulates p then for each action of p of the form $g \xrightarrow{\lambda} st$, p' contains an action of the form $g \wedge g' \xrightarrow{\lambda || \lambda'} st||st'$. (See below for the precise definition.) In other words, p' has an action corresponding to each action of p. To show that p' is using a detector for an action of p, we require that the witness predicate of that detector be $g \wedge g'$ which is the guard of the corresponding action in p'.

Based on the above discussion, given a timed guarded command of the form $g \xrightarrow{\lambda} st$ of p, its (weakest) detection predicate sf and the corresponding action $g \wedge g' \xrightarrow{\lambda || \lambda'} st||st'$ of p', we require that the detection predicate of the desired detector be $g \wedge sf$ and the witness predicate of the desired detector be $g \wedge g'$.

Definition 4.10 **(encapsulates)** We say that p' *encapsulates p from S* iff each action in p' that is enabled in a state in S and that updates variables of p is of the form $g \wedge g' \xrightarrow{\lambda || \lambda'} st||st'$, where $g \xrightarrow{\lambda} st$ is an action of p and st' does not update variables of p and λ' does not include clock variables that are reset in p. Furthermore, an action of the form $g \wedge g' \xrightarrow{\lambda || \lambda'} st||st'$, is executed only when its guard, $g \wedge g'$, is true, and to execute this action st and st' are atomically executed. ∎

In order to show the necessity of existence of detectors, we use the definition of detection predicates and program encapsulation. This necessity applies for the case where the fault-tolerant program is obtained by *reusing* the fault-intolerant program. With this intuition, we next define what we mean by a program reusing another program.

Definition 4.11 **(reuses)** We say that p' *reuses p from S* iff the following two conditions are satisfied:

- p' refines p from S, and
- p' encapsulates p from S. ∎

Based on Definitions 4.10 and 4.11, and applying the concept of weak δ-correctors, we are now ready to show that if a program refines a safety specification $SPEC_{bt} \cap SPEC_{br}$ then it contains detectors and weak δ-correctors. The intuition is that if program p' is designed by transforming p so as to satisfy $SSPEC_{bt}$ (respectively, $SSPEC_{br}$), then the transformation must have added a detector (respectively, a weak δ-corrector) for p, and p' reuses p. We formulate this in Claim 4.12.

Claim 4.12 *Programs that refine a safety specification contain detectors and weak δ-correctors. That is, if*

- *p' reuses p from S, and*
- *p' refines $SSPEC_{bt} \cap SSPEC_{br}$ from S*

then

- *($\forall ac \mid ac$ is an action of p : p' contains a detector of a detection predicate of ac), and*
- *($\forall i \mid 1 \leq i \leq m$: there exists $\delta \in \mathbb{Z}_{\geq 0}$ such that p' contains a weak δ-corrector for the response predicate of the stable bounded response property $P_i \mapsto_{\leq \delta_i} Q_i$ of $SPEC_{br}$).*

Proof. We distinguish two cases based on refinement of $SPEC_{bt}$ and $SPEC_{br}$:

1. (*existence of detectors*) Let ac be an action of p. We show that p' contains a detector of a detection predicate of ac. Let sf be the weakest detection predicate for ac. Since p' reuses p from S, if ac is of the form $g \xrightarrow{\lambda} st$, p' contains an action, say ac', of the form $g \wedge g' \xrightarrow{\lambda || \lambda'} st || st'$.

   Let $Z = g \wedge g'$, and
   $X = g \wedge sf$

   Since $X \Rightarrow sf$, whenever X is true, execution of ac maintains $SSPEC_{bt}$. It follows that X is a detection predicate of ac. We now show that p' refines 'Z detects X' from S.

   - By definition of Z, $Z \Rightarrow g$. Since p' refines $SSPEC_{bt}$ from S, whenever ac is executed in a state where S is true, its execution is safe. Since sf is the weakest detection predicate of ac, $S \wedge Z \Rightarrow sf$. Thus, Safeness is satisfied.
   - Consider any computation, say $\overline{\sigma}'$, of p' which starts in a state where S is true and X is true in each state in $\overline{\sigma}'$. By definition of X, g is true in each state in $\overline{\sigma}'$. Now, consider the computation, say $\overline{\sigma}$, obtained by projecting $\overline{\sigma}'$ on p. Since p' refines p from S, $\overline{\sigma}$ is a computation of p. In $\overline{\sigma}$, g is continuously true. Therefore, by fairness (cf. Assumption 4.5), action ac must eventually be executed. Let σ denote the state where action ac executes in $\overline{\sigma}$, and let σ' denote the corresponding state in $\overline{\sigma}'$. Consider the action ac' executed by p' in state σ'. Since we assume that the transitions included in different guarded commands are mutually exclusive, ac' is the only action that can be executed at σ'. In other words, no other action in p' has the same effect on variables of p from state σ. Thus, Z is true in state σ' and, hence, Progress is satisfied.
   - If Stability does not hold, it implies that even if X is continuously true, the computation does not have a suffix where Z is not continuously true. If Stability does not hold then there exists a computation of p' where the action ac' is never executed. Now, consider this computation of p' and its projection on p. In the projected computation, g is continuously true. However, action ac is not executed. And, this is a contradiction since p' reuses p from S.

2. (*existence of weak δ-correctors*) As mentioned in Section 2, $SPEC_{br}$ is a conjunction of $m \in \mathbb{Z}_{\geq 0}$ stable bounded response properties. We show that for each such property of the form $P_i \mapsto_{\leq \delta_i} Q_i$, where $1 \leq i \leq m$, p' contains a weak δ-corrector.

Let $X = Q_i$, and

$Z = Q_i \wedge \{\sigma \mid \sigma$ is a state of p' such that σ is reached in some computation of p' starting from $P_i \wedge S\}$.

Now, we show that there exists $\delta \in \mathbb{Z}_{\geq 0}$ such that p' refines 'Z weakly corrects X within δ' from S:

- By definition of Z, in any state where Z is true, Q_i is true. Thus, Safeness is satisfied.
- Since p' refines $P_i \mapsto_{\leq \delta_i} Q_i$ from S every computation of p' starting from $S \wedge P_i$ will reach a state where $S \wedge Q_i$ is true within δ_i. By definition of Z, Z is true in this state as well. Thus, Progress is satisfied.
- By the same token and choosing $\delta = \delta_i$, Weak Convergence within δ is satisfied.
- Since p' refines p from S, it follows that S is closed in p'. Obviously, the second conjunct in Z is closed in p' as well. Thus, Z is closed in p' and, hence, Stability is satisfied. ∎

We now use Claim 4.12 to show that if a hard-failsafe f-tolerant program p' is designed by reusing p then p' contains a hard-failsafe tolerant detector for each action of p and a hard-failsafe tolerant weak δ-corrector for each stable bounded response property.

Theorem 4.13 *Hard-failsafe f-tolerant programs contain hard-failsafe tolerant detectors and weak δ-correctors. That is,*

if

- *p refines $SPEC$ from S,*
- *p' reuses p from R, where $R \Rightarrow S$, and*
- *p' is hard-failsafe f-tolerant to $SPEC_{bt} \cap SPEC_{br}$ from R*

then

- *($\forall\ ac \mid ac$ is an action of p : p' is a hard-failsafe f-tolerant detector of a detection predicate of ac), and*
- *($\forall i \mid 1 \leq i \leq m$: there exists $\delta \in \mathbb{Z}_{\geq 0}$ such that p' contains a hard-failsafe f-tolerant weak δ-corrector for the response predicate of stable bounded response property $P_i \mapsto_{\leq \delta_i} Q_i$ of $SPEC_{br}$).*

Proof. Similar to Claim 4.12, we distinguish two cases based on refinement of $SPEC_{bt}$ and $SPEC_{br}$:

1. (*existence of hard-failsafe tolerant detectors*) Let sf be the weakest detection predicate for ac. Since p' encapsulates p from R, if ac is of the form $g \xrightarrow{\lambda} st$, p' contains an action, say ac', of the form $g \wedge g' \xrightarrow{\lambda || \lambda'} st || st'$.

   Let $Z = g \wedge g'$, and let
   $X = g \wedge sf$

   Since $X \Rightarrow sf$, whenever X is true, execution of ac maintains $SPEC_{bt}$. It follows that X is a detection predicate of ac. We now show that p' is hard-failsafe f-tolerant for 'Z detects X' from R. To this end, we first show that p' refines 'Z detects X' from R. Then, we show that there exists an f-span T such that $p'[]f$ refines the hard-failsafe tolerance specification of 'Z detects X' from T:

- For the first part, since $R \Rightarrow S$, we observe that p' refines $SSPEC_{bt}$ from R. Therefore, as in Claim 4.12, it follows that p' refines 'Z detects X' from R.
- For the second part, we let T be the f-span used to show that p' is hard-failsafe f-tolerant for $SPEC_{bt}$ from R. In this part, we need to show that a computation of $p'[]f$ satisfies Safeness and Stability. This proof is identical to the first part of proof of Safeness and Stability in Claim 4.12.

2. (*existence of hard-failsafe tolerant weak δ-correctors*) We show that for each stable bounded response property of the form $P_i \mapsto_{\leq \delta_i} Q_i$, where $1 \leq i \leq m$, there exists $\delta \in \mathbb{Z}_{\geq 0}$ such that p' contains a hard-failsafe tolerant weak δ-corrector for Q_i.

   Let $X = Q_i$, and

   $Z = Q_i \wedge \{\sigma \mid \sigma$ is a state of p' such that σ is reached in some computation of p' starting from $P_i\}$.

   Now, we show that there exists $\delta \in \mathbb{Z}_{\geq 0}$ such that p' is hard-failsafe f-tolerant for 'Z weakly corrects X within δ'. To this end, we first show that p' refines 'Z weakly corrects X within δ' from R. Then, we show that there exists an f-span T such that $p'[]f$ refines the hard-failsafe tolerance specification of 'Z weakly corrects X within δ' from T:

   - For the first part, since $R \Rightarrow S$, we observe that p' refines $SSPEC_{br}$ from R. Therefore, as in Claim 4.12, it follows that there exists $\delta \in \mathbb{Z}_{\geq 0}$ (namely $\delta = \delta_i$) such that p' refines 'Z weakly corrects X within δ' from R.
   - For the second part, we let T be the f-span used to show that p' is hard-failsafe f-tolerant for $SPEC_{br}$ from R. In this part, we need to show that a computation of $p'[]f$ satisfies Safeness, Stability, and Weak Convergence. The proof of Safeness and Stability is identical to the second part of proof of Claim 4.12. In order to show that a computation of $p'[]f$ satisfies Weak Convergence, first, recall that by Assumption 2.14, at most k faults may occur. Hence, p' reaches a state where Q_i is true within at most $k.\delta_i$ time units. Thus, by choosing $\delta' = |f^s|.\delta_i + \sum f^t$ p' is hard-failsafe f-tolerant to 'Z weakly corrects X within δ' from R. ∎

4.2.1 Example

We now illustrate an instance of detectors and weak δ-correctors in hard-failsafe programs using an altitude switch example from [8]. The altitude switch program (ASW) reads a set of input variables coming from an altitude meter. Then, it powers on a Device of Interest (DOI) when the aircraft descends below a pre-specified altitude threshold. There exist six internal variables, a mode variable that determines the operating mode of the program, three watchdog timers, and an input variable that represents the state of the altitude meter:

- The internal variables are as follows: (i) *mAltBelow* is equal to 1 if the altitude is below a pre-specific threshold, otherwise, it is equal to 0; (ii) *mDOIStatus* is equal to 1 if the DOI is powered on, otherwise, it is equal to 0; (iii) *mInit* is equal to 1 when the system is being initialized; otherwise, it is equal to 0; (iv) *mInhibit* is equal to 1 when the DOI power-on is inhibited; otherwise, it is equal to 0, (v) *mReset* is equal to 0 if a system reset has been initiated. Otherwise, it is equal to 1, and (vi) *iCorruptSensor* is equal to 1 if the system reads an invalid value from the altitude meter sensors. Otherwise, it is equal to 0.

- The ASW program can be in three different modes: (i) initialization mode when the system is initializing; (ii) await mode when the system is waiting for the DOI to power on, and (iii) standby mode. We use the variable $mcStatus$ with domain $\{\textsc{Init}, \textsc{Await}, \textsc{Standby}\}$ to show the system modes in the program.
- We model the signals that come from the altitude meter by the variable $iSensorReading$. This variable is equal to $\perp$ when the system fails to read the altitude meter. Otherwise, it is not equal to $\perp$.
- We introduce three watchdog timers: (i) the clock variable x measures the time elapsed since the system has been in the INIT mode; (ii) the clock variable y measures the time elapsed since the system has failed to read the altitude meter (i.e., $iCorruptSensor = 1$), and (iii) the clock variable z measures the time elapsed since the system has been in the AWAIT mode.

Fault-intolerant program. The program ASW changes its mode from INIT to STANDBY within 1 second (Action ASW1 in the program below). Also, the program may go back to the INIT mode if it is in STANDBY mode and the reset signal is received (Action ASW2). If the program is in the STANDBY mode, the DOI power-on is not inhibited, and the DOI is not powered on then the program *may* go to a state where the altitude of the aircraft is below the threshold and it is in the AWAIT mode (Action ASW3). In this case, the program starts timer z. Otherwise, the program stays in the STANDBY mode for an arbitrary time as long as the altitude meter is not showing an invalid value (Action ASW6). In the AWAIT mode, the program either (i) powers on the DOI within 2 seconds and goes back to the STANDBY mode (Action ASW4), or (ii) goes to the INIT mode upon receiving the reset signal (Action ASW5). The program may take delays as long as as it does not violate the timing constraints of the program (Action ASW6). If the program receives a signal indicating that the altitude meter sensors are reading an invalid value, it sets the variable $iCorruptSensor$ to 1 and starts the timer y (Action ASW8). In this case, the program should go back to the INIT mode within 2 seconds (Action ASW9).

$$
\begin{array}{lcl}
\textsc{Asw}1 :: (mcStatus = \textsc{Init}) \wedge (mInit = 1) \wedge (x \leq 1) & \longrightarrow & mcStatus, mInit := \textsc{Standby}, 0 \\
\textsc{Asw}2 :: (mcStatus = \textsc{Standby}) \wedge (mReset = 0) & \longrightarrow & mcStatus, mReset := \textsc{Init}, 1 \\
\textsc{Asw}3 :: (mcStatus = \textsc{Standby}) \wedge (mAltBelow = 0) \wedge & & \\
\qquad\qquad (mInhibit = 0) \wedge (mDOIStatus = 0) & \xrightarrow{\{z\}} & mcStatus, mAltBelow := \textsc{Await}, 1 \\
\textsc{Asw}4 :: (mcStatus = \textsc{Await}) \wedge (mDOIStatus = 0) \wedge (z \leq 2) & \longrightarrow & mcStatus, mDOIStatus := \textsc{Standby}, 1 \\
\textsc{Asw}5 :: (mcStatus = \textsc{Await}) \wedge (z \leq 2) & \xrightarrow{\{x\}} & mcStatus, mReset := \textsc{Init}, 1 \\
\textsc{Asw}6 :: ((mcStatus = \textsc{Standby}) \wedge (iCorruptSensor \neq 1)) \vee & & \\
\qquad\qquad (x \leq 1) \vee (y \leq 2) \vee (z \leq 2) & \longrightarrow & \textbf{wait} \\
\textsc{Asw}7 :: (mInit = 0) & \xrightarrow{\{x\}} & mInit = 1 \\
\textsc{Asw}8 :: (iSensorReading = \perp) \wedge (iCorruptSensor = 0) & \xrightarrow{\{y\}} & iCorruptSensor = 1 \\
\textsc{Asw}9 :: (mcStatus = \textsc{Standby}) \wedge (iCorruptSensor = 1) \wedge (y \leq 2) & \longrightarrow & mcStatus, iCorruptSensor := \textsc{Init}, 0
\end{array}
$$

The invariant of the ASW program consists of the following states:

$$S_{\text{Asw}} = \{\sigma \mid ((\mathit{mcStatus}(\sigma) = \textsc{Init}) \Rightarrow (x(\sigma) \leq 1)) \wedge$$
$$((\mathit{mcStatus}(\sigma) = \textsc{Standby}) \Rightarrow ((\mathit{iCorruptSensor}(\sigma) = 0) \vee (y(\sigma) \leq 2))) \wedge$$
$$((\mathit{mcStatus}(\sigma) = \textsc{Await}) \Rightarrow (z(\sigma) \leq 2))\}.$$

The safety specification requires that the program does not change its mode from STANDBY to AWAIT, if it fails to read the altitude meter. Also, when the state of the program is perturbed, it can only go to the INIT mode. Furthermore, the safety specification has three bounded response properties that specifies the timing constraints of the altitude switch. Formally, the safety specification is as follows:

$$\mathit{SPEC}_{bt} = \{(\sigma_0, \sigma_1) \mid ((\mathit{iCorruptSensor}(\sigma_0) = 1) \wedge (\mathit{mcStatus}(\sigma_0) = \textsc{Standby}) \wedge (\mathit{mcStatus}(\sigma_1) = \textsc{Await})) \vee$$
$$((\sigma_0 \notin S_{\text{Asw}}) \wedge ((\mathit{mcStatus}(\sigma_1) = \textsc{Standby}) \vee (\mathit{mcStatus}(\sigma_1) = \textsc{Await}))) \vee$$
$$((\sigma_0 \notin S_{\text{Asw}}) \wedge (\mathit{mReset}(\sigma_1) = 1))\}$$

$$\mathit{SPEC}_{br_1} = ((\mathit{mcStatus} = \textsc{Init}) \mapsto_{\leq 1} (\mathit{mcStatus} = \textsc{Standby})) \wedge$$
$$((\mathit{mcStatus} = \textsc{Await}) \mapsto_{\leq 2} ((\mathit{mcStatus} = \textsc{Standby}) \wedge (\mathit{mDOIStatus} = 1)) \vee (\mathit{mcStatus} = \textsc{Init}))$$

$$\mathit{SPEC}_{br_2} = ((\mathit{mcStatus} = \textsc{Standby}) \wedge (\mathit{iCorruptSensor} = 1)) \mapsto_{\leq 2} (\mathit{mcStatus} = \textsc{Init})$$

Fault actions. The ASW program is subject to a set of delay faults when the program is (i) initializing; (ii) in a state where altitude meter shows invalid values, or (iii) is waiting for the DOI to power on. Moreover, faults can corrupt the reading of the altitude meter as well.

$$F1 :: (\mathit{mcStatus} = \textsc{Init}) \wedge (\mathit{mInit} = 1) \wedge (1 \leq x \leq 2) \longrightarrow \textbf{wait}$$
$$F2 :: (\mathit{mcStatus} = \textsc{Standby}) \wedge (\mathit{iCorruptSensor} = 1) \wedge (2 \leq y \leq 3) \longrightarrow \textbf{wait}$$
$$F3 :: (\mathit{mcStatus} = \textsc{Await}) \wedge (2 \leq z \leq 3) \longrightarrow \textbf{wait}$$
$$F4 :: (\mathit{iSensorReading} \neq \bot) \longrightarrow \mathit{iSensorReading} := \bot$$

Hard-failsafe fault-tolerant program with respect to SPEC_{bt} and SPEC_{br_2}. Observe that in the intolerant program, action ASW3 does not maintain the safety specification when the altitude meter shows invalid value (i.e., fault $F4$ occurs). To preserve the safety specification, we will use a detector DR. It is easy to see that in DR the detection predicate $X = (\mathit{mcStatus} = \textsc{Standby}) \wedge (\mathit{iCorruptSensor} \neq 1)$ and the witness predicate $Z = \mathit{Guard}(\textsc{Asw}3) \wedge (\mathit{iCorruptSensor} \neq 1)$. Thus, to add hard-failsafe fault-tolerance, ASW3 is restricted to execute only when the witness predicate of DR is true. Thus, we have

$$\textsc{Hfs\_Asw}3 :: (\mathit{mcStatus} = \textsc{Standby}) \wedge (\mathit{mAltBelow} = 0) \wedge (\mathbf{iCorruptSensor \neq 1}) \wedge$$
$$(\mathit{mInhibit} = 0) \wedge (\mathit{mDOIStatus} = 0) \xrightarrow{\{z\}} \mathit{mcStatus}, \mathit{mAltBelow} := \textsc{Await}, 1$$

Moreover, we require that if the program fails to read the altitude meter, it should initialize in at most 2 seconds even in the presence of the delay fault $F2 :: (\mathit{mcStatus} = \textsc{Standby}) \wedge (\mathit{iCorruptSensor} = 1) \wedge (2 \leq y \leq 3) \rightarrow$ **wait**. To preserve this property, we will use a weak δ-corrector. In particular, actions ASW6 and ASW9 can potentially be such a corrector. Thus, it suffices to strengthen action ASW6 to get a weak 1-corrector that tolerates $F2$ by allowing delay transitions only for at most 1 second when the program is in the STANDBY mode and the altitude meter reads a corrupted value:

$$\text{HFS\_ASW6} :: ((mcStatus = \text{STANDBY}) \wedge (iCorruptSensor \neq 1)) \vee$$
$$(x \leq 1) \ \vee (\mathbf{y} \leq \mathbf{1}) \ \vee \ (z \leq 2) \qquad \longrightarrow \qquad \textbf{wait}$$

Observe that Actions ASW6 and ASW9 in the intolerant program already provide the required computations to meet the deadline of reaching the STANDBY mode in the fault-span. Thus, in the design of the weak δ-corrector, we did not add any "recovery" computations to the tolerant program. In fact, in order to design the weak δ-corrector, we removed the computations that do not meet the deadline in the presence of faults by strengthening the Action ASW6.

4.3 The Role of Real-Time Fault-Tolerance Components in Soft and Hard-Masking Programs

In this subsection, we present the theory of necessity of real-time fault-tolerance components in masking tolerant programs. We distinguish three levels of masking fault-tolerance, namely, (1) soft-masking with bounded-time recovery, (2) hard-masking with bounded-time recovery, and (3) hard-masking with unbounded-time recovery. Recall that necessity of existence of components in soft-masking tolerance with unbounded-time recovery has been shown in [1]. For reasons space and similarity of proofs to previous sections, in this subsection, we only outline the main theorems and we refer the reader to `http://www.cse.msu.edu/~borzoo/proofs.pdf` for the detailed proofs.

Soft-masking with bounded-time recovery. Since a soft-masking program must maintain $SPEC_{bt}$ in the presence of faults and it guarantees bounded-time recovery, we can decompose it into a fault-intolerant program, detectors to refine $SPEC_{bt}$, and strong δ-correctors to provide bounded-time recovery.

Theorem 4.14 *Soft-masking f-tolerant programs with recovery time θ contain soft-masking tolerant detectors and strong δ-correctors. That is,*
if

- *p refines SPEC from S,*
- *p' reuses p from R, where $R \Rightarrow S$,*
- *p' refines $SPEC_{bt \leq \theta} p | R$ from T, where $T \Leftarrow R$, and*
- *p' is soft-masking f-tolerant to SPEC from T,*

then

- *($\forall$ ac | ac is an action of p : p' is a soft-masking f-tolerant detector of a detection predicate of ac),*
- *there exists δ such that p' is a soft-masking tolerant strong δ-corrector of an invariant predicate of p, and*
- *p' is a nonmasking f-tolerant δ-corrector with recovery time θ' of an invariant predicate of p for some θ'.* ∎

Hard-masking with bounded-time recovery. Since a hard-masking program must maintain both $SPEC_{bt}$ and $SPEC_{br}$ in the presence of faults and it guarantees bounded-time recovery, we can decompose it into a fault-intolerant program, detectors for refining $SPEC_{bt}$, weak δ-correctors for refining $SPEC_{br}$ and strong δ-correctors to provide bounded-time recovery.

Theorem 4.15 *Hard-masking f-tolerant programs with recovery time θ contain hard-masking tolerant detectors, weak and strong δ-correctors. That is,*
if

- *p refines SPEC from S,*
- *p' reuses p from R, where $R \Rightarrow S$,*
- *p' becomes $p|R$ within θ from T, where $T \Leftarrow R$, and*
- *p' is hard-masking f-tolerant to SPEC from T,*

then

- *($\forall\, ac$: ac is an action of p : p' is a hard-masking f-tolerant detector of a detection predicate of ac),*
- *($\forall i : 0 \leq i \leq m$: there exists $\delta \in \mathbb{Z}_{\geq 0}$ such that p' contains a hard-masking f-tolerant weak δ-corrector for the response predicate of stable bounded response property $P_i \mapsto_{\leq \delta_i} Q_i$ of $SPEC_{br}$),*
- *there exists δ' such that p' is a hard-masking tolerant strong δ'-corrector of an invariant predicate of p, and*
- *p' is a nonmasking f-tolerant δ'-corrector with recovery time θ' of an invariant predicate of p for some θ'.* ∎

Observe that in case of unbounded-time soft-masking (respectively, hard-masking) programs, in Theorem 4.14 (respectively, Theorem 4.15), we only need to replace the strong δ-correctors (respectively, strong δ'-correctors) by strong ∞-correctors. The rest of the statement of both theorems and proofs remain unchanged.

4.4 Example

We now illustrate the role of strong δ-correctors in hard-masking fault-tolerance in the ASW program presented in Subsection 4.2.1. More specifically, we enhance fault-tolerance level of ASW from hard-failsafe to hard-masking with recovery time $\theta = 1$. Using the hard-failsafe program in Subsection 4.2.1, it only remains to add a strong δ-corrector that provides recovery to the invariant. Notice that according to $SPEC_{bt}$ of ASW, recovery is only possible to the states where the program is in the INIT mode, which (similar to the alternating bit protocol) in turn identifies the witness predicate. Thus, the actions of the strong δ-corrector is as follows:

```
$CR1$ :: $(1 \leq x \leq 2) \vee (2 \leq z \leq 3) \vee$
         $((2 \leq y \leq 3) \wedge (iCorruptSensor = 1))$    $\xrightarrow{\{x,y,z\}}$    $mcStatus, mReset := \text{INIT}, 1$
$CR2$ :: $(1 \leq x \leq 2) \vee (2 \leq z \leq 3) \vee$
         $((2 \leq y \leq 3) \wedge (iCorruptSensor = 1))$    $\longrightarrow$    **wait**
```

5 Conclusion and Future Work

In this paper, we focused on a theory of fault-tolerance components in the context real-time programs. We showed that these components, namely, detectors and weak/strong δ-correctors, are integral parts of fault-tolerant real-time programs that reuse their fault-intolerant version. In particular, we showed that depending upon the level of fault-tolerance, namely *soft-failsafe, hard-failsafe, nonmasking, soft-masking*, and *hard-masking*, the fault-tolerant real-time program contains one or more of such components.

We note that the use of detectors and correctors from [1] in untimed programs has been illustrated in terms of several examples such as repetitive Byzantine agreement, mutual exclusion, tree maintenance, leader election, termination detection, bounded network management, etc. We are currently investigating the use of detectors and weak/strong δ-correctors for developing design methods for fault-tolerant real-time programs.

We are also focusing on using these components in the context of automated addition of fault-tolerance. In particular, by designing and verifying these components in advance, it would be possible to speed up automated addition of fault-tolerance to real-time programs. Towards this end, we are working on extending our current work on automated addition of fault-tolerance (cf. `http://www.cse.msu.edu/~sandeep/publications`) to use detectors and weak/strong δ-correctors.

References

[1] A. Arora and S. S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 436–443, 1998.

[2] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[3] T. A. Henzinger. Sooner is safer than later. *Information Processing Letters*, 43(3):135–141, 1992.

[4] R. Alur and T. A. Henzinger. Real-time system = discrete system + clock variables. *International Journal on Software Tools for Technology Transfer*, 1(1-2):86–109, 1997.

[5] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[6] A. Arora and M. G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.

[7] B. Bonakdarpour and S. S. Kulkarni. Incremental synthesis of fault-tolerant real-time programs. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 122–136, 2006.

[8] R. Bharadwaj and C. Heitmeyer. Developing high assurance avionics systems with the SCR requirements method. In *19th Digital Avionics Systems Conference*, 2000.